Aspects and Modules Combined

Johan Ovlinger Karl Lieberherr David H. Lorenz

College of Computer Science
Northeastern University
Boston, Massachusetts 02115-5000, USA
{johan,lieber,lorenz}@ccs.neu.edu

ABSTRACT

Overly regular module interfaces in object-oriented languages hamper modularization of complex applications. Aspect-oriented programming tackles this problem by allowing module boundaries to span and partition classes in a flexible manner. However, not without a cost. In order to achieve this flexibility, common modularity mechanisms, such as encapsulation and external composition, are lost. The ability to separately compile or reason about a modular unit is also compromised.

Combining aspects and modules restores these properties to the aspect-oriented programming language. In restoring the properties, the programming units—Aspectual Collaboration in our case—become more verbose: Encapsulation requires that all collaborations describe their interface to the rest of the application; and composition of collaborations similarly requires that each module's interface be reconciled with the other. We give a brief introduction to Aspectual Collaborations, however, the main part of the paper covers a comparison study of AspectJ, Hyper/J, and Aspectual Collaborations in solving an AOP programming challenge. We derive the comparative cost of using encapsulation.

1. INTRODUCTION

Separation of concerns and *modularity* [36] are at the heart of the programming process [11]. Concerns are conceptual units in which one decomposes a given problem. Modules are software units in which one organizes software. Modular programming is conducted by expressing programming concerns in modular units.

The key insight of *Aspect-Oriented Programming* (AOP) [23] is that module boundaries seldom fit along *all* concern boundaries. The modular breakdown captures some programming concerns, while other concerns, namely, cross-cutting concerns, are scattered throughout the units of modularity resulting in tangling of the concerns' code. (For a survey of AOP techniques see [13].)

In principle, AOP alleviates the problem of scattering and tangling by separately expressing each cross-cutting concern in terms of its own modular structure, as an external aspectual unit. The aspectual unit can then be re-attached, across modular boundaries, to the host application. In practice, however, the two leading AOP projects and de facto standards, AspectJ [5] and Hyper/J [39], exemplify a tradeoff between flexibility and structure.

On one hand, following the tradition of meta-object protocols [24] and open implementations [22], AspectJ offers programming flexibility with minimal structural constraints. Constraints are mostly enforced by good programming practices rather than language abstractions. AspectJ discriminates between methods, which capture base behavior, and aspectual units, which affect the base behavior. Aspects can affect the base behavior, however, methods cannot affect the aspectual unit.

Hyper/J, on the other hand, emphasizes structured software engineering rather than aspectual features. Rooted in the subject-oriented programming paradigm [17], Hyper/J supports merging and decomposition of separately specified class hierarchies. Unlike AspectJ, Hyper/J treats the base and the aspectual behaviors symmetrically, that is, a hyperslice can model both the base and the aspectual unit.

The two approaches seem to suggest an aspectual spectrum for AOP languages. Table 1 shows this spectrum, and summarizes the conclusion we draw from a comparison of Aspectual Collaboration [26], Hyper/J and AspectJ in the latter part of the paper. Justification and discussion of judgments are delayed till after the comparison.

1.1 Objective

The goal of this work is achieving a balance between *flexibility* and *structure* in an aspect-oriented programming approach. By flexibility, we refer to supporting conventional reuse of concerns, such as inheritance and genericity, in a wide variety of contexts. Structure refers to static properties that aid program comprehension, such as encapsulation and type-safety.

In this paper, we provide a novel means for consolidating the two currently separate stands of aspect-oriented work. We show that structured units and flexible attachments are not necessarily mutually exclusive properties. We illustrate concretely an aspectual language, namely Aspectual Collaborations, in which the aspectual units are both structured and flexible.

The approach is shown to be more structural than AspectJ and have more flexibility than Hyper/J. The paper focuses on a comparison of Aspectual Collaborations with AspectJ and Hyper/J, while the prototype implementation

Weaver	AspectJ	Aspectual Collaborations	Hyper/J
Emphasis	Programming	Components	Software engineering
Approach	Aspectual	Composable	Separational
Aspect	Aspect class	Collaboration	Hyper-slice
Aspect Hook	Join point	Join graph	Hyper-module
Attachment	Point-cut designator	Graph pattern matching	Hyper-module
Structure	-	+	+
Flexibility	+	+	-
Symmetry	-	+	+

Table 1: The aspectual spectrum

is discussed elsewhere [26].

1.2 Motivation

An aspectual interface specifies the contract enforced between the aspectual unit and the host program at every attachment. The aspectual interface controls not only how the host program sees the aspectual unit and how it sees the host program, but also what behavior is expected and what is provided at each attachment occurrence.

The interface must balance the need for well-definedness against the need for flexibility. For example, the trivially complete interface (requiring a copy of the host application) surely allows rich behavior to be woven in, but severely restricts where and how the aspectual unit can be attached. The trivially empty interface, on the other hand, provides ample opportunities for reuse, but offers no interface for behavior. The interface must thus be structured enough so that interesting aspects can be written against it on the aspectual unit side, yet flexible enough to accommodate a wide variety of host programs on the application side.

An immediate benefit of a well defined aspectual interface is the possibility to *separately compile* aspectual units before composing them. The aspectual interface allows the weaver compiler to analyze aspectual units and the host program in isolation, and then to derive the meaning of the combined whole by additionally analyzing how they have been attached together. This baseline for how much analysis is needed to enable compilation of aspectual interfaces influences where the balance between flexibility and analysis lands.

The difficulty is to retain some measure of generality in the aspectual behavior. For example, an Aspectual Collaboration for the standard logging aspect should be applicable to methods of any signature, yet still have an interface specific enough to enable separate compilation.

The novelty in Aspectual Collaboration lies in the following characteristics:

- *Encapsulation.* The aspectual units emphasize encapsulation above everything else. At all stages, Aspectual Collaborations are encapsulated behind interfaces. These interfaces define the requirements an Aspectual Collaboration makes of contexts where it is to be used, and what parts of itself it is willing to let external behavior see. As a result, separate compilation of aspectual units is made possible. The strong encapsulation interface also allows an Aspectual Collaboration to be separately understood. For example, an Aspectual Collaboration can guarantee that a method never be advised by not exporting it.

- *External composition.* Aspectual Collaborations are composed with other Aspectual Collaborations externally. Attachment (as we refer to composition) reconciles all name differences between the constituent Aspectual Collaborations, along with all imports and exports, to generate a new Aspectual Collaboration. Only those parts of the constituent Aspectual Collaborations explicitly exported are visible outside the composition. Successful composition guarantees that the composed Aspectual Collaboration is no less type safe than the constituents in isolation.

  Composition is also how aspectual (cross-cutting) features are attached to the join-points they advise, allowing us to apply the same encapsulation and visibility semantics to aspectual behavior as we do to modular features. This is a very powerful feature, as it enables the programmer to reason about an aspectual program in a *compositional* manner, easily understanding exactly how the various modules cross cut each other.

- *Generic, type-safe advice.* A novel API allows completely type-safe and around methods, while maintaining full generic applicability (can wrap methods of any signature) and control over the `proceed` call, and not requiring any down casts by the programmer. Optionally, user controlled fractions of genericity can be traded in for more control of the details of an intercepted method call. Our implementation leverages encapsulation and Java's type checks to guarantee that such generic methods are type safe.

- *Small, orthogonal language.* We purposefully keep our core language minimal, for example eschewing "before" and "after" methods for the more general "around". This allows us to keep our implementation–and more importantly, our semantics–small and easily understood. This allows the user to predict what is going to happen when a language feature is used in a non-standard way, and more importantly to understand why something failed to work when it goes wrong. We call this the "wizard" factor.

1.3 Roadmap

Section 2 allows the reader to understand the main idea of Aspectual Collaborations on a general level though simple examples. Section 3 introduces a small concrete programming challenge. The programming challenge examples provide a level comparison benchmark, in that each approach is then asked to solve the same problem in a reusable style. Sections 4, 5, and 6 that follow compare solutions using Aspectual Collaborations, Hyper/J, and AspectJ, respectively.

This serves as a preliminary validation of Aspectual Collaborations, showing in Section 7 that they compare favorably with the other two systems in ease of use and expressiveness, while providing a higher degree of static safety and reuse. Related and future works are brought in Section 8. Section 9 concludes.

2. ASPECTUAL COLLABORATIONS OUTLINED

An Aspectual Collaboration has the following ingredients:

- A **collaboration** is an intrinsically named graph of *participants*, where the nodes are formal classes, and the edges are *is-a* and *has-a* relations and functional relations.

- Each participant has zero or more *members*. A member is a field or a method, obeying the the normal semantics of Java. [1] In addition to the member modifiers offered by Java, participants can have three additional orthogonal member modifiers:

  - **expected** members are deferred members that must be provided before the collaboration can be executed. These differ from **abstract** members in that both methods and fields can be expected, and more importantly that an expected method does not inhibit class instantiation.

  - **aspectual** methods are able to intercede in invocations of other methods. When an aspectual method is attached to a host method, any invocation of the host method is intercepted, reified as a method call object, and passed to the aspectual method as an argument. The aspectual method can then control and modify the details of the call, until returning control to the original caller. An aspectual method may or may not be constrained to work with only a more limited set of signatures in order to gain functionality.

  - **exported** members are visible outside the collaboration. Export modifiers wear off after each collaboration composition, so are not specified at the member definition, but externally to the member participant. In other respects, it is a normal member-specific modifier.

- *Required and provided interfaces* are derived from the member declarations on the participants of a collaboration. A member is often in both interfaces.[2] Each member implicitly knows which participant it is defined on, so we define the interfaces in terms of members.

  - The *required interface* consists of all **expected** members. It declares holes in the implementation which need to be provided to complete the collaboration.

  - The *provided interface* consists of all **exported** members. The provided interface allows the behaviors of the Aspectual Collaboration to be accessed from the outside.

[1] We use the terms *member* and *field* to mean *properties* and *attributes*, respectively.

[2] The provided interface must include all unprovided expected members, as otherwise an expected member becomes unexported and cannot be referenced from outside the collaboration, making it impossible to ever provide an implementation for it. The collaboration is then doomed to never be executable, so at each composition, an expected method must be provided or exported.

Listing 1: A simple host package

```
1 package variables;
2 class Vars {
3   String foo;
4   Baz bar;
5 }
6 class Baz {
7   Vars var;
8 }
```

An Aspectual Collaboration definition has the following syntactic structure:

collaboration *name*
{ **extends** *collaboration* }*
{ **participant** *formal_class_def* }*
{ [**match** *roles*] **attach** *collaboration* }*

A simple Aspectual Collaboration, without the optional **extends** and **match**–**attach** clauses, has the same shape as a Java package, substituting **collaboration** for **package** and **participant** for **class**. In fact, collaborations are a superset of packages,[3] and a package is accepted as a (grounded) collaboration even without substituting the keywords. Thus, the host package variables (Listing 1) is a legal collaboration.

The **extends** clause imports another collaboration, reexporting all visible members and classes. This is exactly the equivalent of manually declaring empty participants, attaching the extended collaboration to them, and exporting all visible members, but the operation is common enough to warrant direct language support. The **match** clause supports convenient specification of multiple attachments of collaborations (analogous to AspectJ pointcuts), but will not be used in this paper, as all collaborations will be attached exactly once.

Unlike a package, however, a collaboration is generally a specification with "holes": a participant can reference expected features (attributes and methods) as if they were defined. In adviceSetGetAttribute (Listing 2), the HasAttribute participant defines a pair of `set` and `get` methods for an expected attribute, which will be provided only later. The double braces around the method bodies should be read as if they were a single brace. Their sole purpose is to allow our implementation to avoid parsing full Java. The **limited** keyword allows for a "frozen" participant (such as String and `int`), for which all modifications are prohibited, to play the role of AttributeType.

Generally, an Aspectual Collaboration is a formal class graph (a collaboration), which can be superimposed [21] on

[3] The association between collaborations and packages is fundamental to our implementation: An Aspectual Collaboration is separately compiled into `.class` files of a single package.

Listing 2: Defining generic setters and getters

```
1 collaboration adviceSetGetAttribute;
2 participant HasAttribute {
3   expected AttributeType aName;
4   public void set(AttributeType aName) {{
5     this.aName = aName; }}
6   public AttributeType get() {{
7     return aName; }}
8 }
9 limited participant AttributeType;
```

Listing 3: Defining a foo setter getter

```
1 collaboration adviceSetGetFoo;
2 extends variables;
3 attach adviceSetGetAttribute {
4   Vars += HasAttribute {
5     provide aName with foo;
6     export set as set_foo;
7     export get as get_foo;
8   }
9   String += AttributeType;
10 }
```

Listing 4: Attaching a collaboration to itself

```
1 collaboration adviceSetGetTwoAttributes;
2 extends adviceSetGetAttribute {
3   export AttributeType as AttributeTypeA {
4     export set as setA;
5     export get as getA;
6   }
7 }
8 attach adviceSetGetAttribute {
9   adviceSetGetTwoAttributes.HasAttribute +=
10        HasAttribute {
11    export aName as bName;
12    export set as setB;
13    export get as getB;
14  }
15  export AttributeType as AttributeTypeB;
16 }
```

another class graph. We refer to the act of superimposing as *attaching* a collaboration to a host collaboration; the host collaboration is said to have been *decorated*. Expected features declare holes in the formal class graph, while all other features are introduced by the attachment; and aspectual methods can also intercede in the invocation of methods to which they are attached. Expected and aspectual features offer two alternative ways, explicit and implicit, respectively, to transfer control and information from one collaboration to another.

The adviceSetGetAttribute collaboration (Listing 2) defines a formal class graph consisting of two participants, HasAttribute and AttributeType, where the former is expected to have a (direct) reference to the latter. When superimposed on an actual class graph, the adviceSetGetAttribute aspectual collaboration will introduce a setter and a getter for the expected reference. The adviceSetGetFoo collaboration (Listing 3), for example, is created by attaching adviceSetGetAttribute to the contents of the variables package to introduce a setter and getter for the `foo` field of Vars.[4]

A **collaboration** definition incrementally builds an Aspectual Collaboration by first declaring the skeleton collaboration (a set of participants) and then attaching additional collaborations to that skeleton. The skeleton collaboration's participants (a formal class graph) can be directly declared (e.g., HasAttribute and AttributeType in adviceSetGetAttribute) or implicitly acquired from another collaboration using the **extends** keyword (e.g., variables in adviceSetGetFoo). Once built, the skeleton collaboration can be decorated by attaching any number of collaborations to it (e.g., adviceSetGetAttribute is superimposed on variables in adviceSetGetFoo). The final formal class graph is the result of the Aspectual Collaboration declaration.

The += operators in adviceSetGetFoo (Listing 3) emphasize that the attached roles HasAttribute and AttributeType may have a structural and behavioral affect on the local participants Vars and String. Vars is decorated with two methods by the behavior introduced in HasAttribute. In addition to the introduction, the += operator also redirects all references (if any—there are none in this example) from the inserted type, HasAttribute, to the destination type, Vars. String, on the other hand, cannot be decorated at all, but allowed as a *lvalue* of += thanks to the **limited** keyword in adviceSetGetAttribute. However, by redirecting references from AttributeType to String, we achieve type parameterization as a degenerate case of attachment [1, 42].

An Aspectual Collaboration can be extended and attached to any host collaboration, not just a (grounded) package. The construction is associative. When an Aspectual Collaboration is extended, the host collaboration can reuse its formal class graph as if it was defined locally. When an Aspectual Collaboration is attached to a host collaboration, the attached formal class graph is mapped against the host formal class graph. Any expected feature in the attached Aspectual Collaboration needs to be either mapped to a concrete feature, or exported for later mapping. For example, the adviceSetGetTwoAttributes collaboration (Listing 4) first extends adviceSetGetAttribute and then attaches the adviceSetGetAttribute collaboration to itself, resulting in a collaboration that introduces setters and getters for *two* attributes.

[4]In the cases that there are many fields and only some of the fields should be provided with getters and setters a more expressive matching is available.

2.1 Aspectual Methods Revisited

As soon as a host method with aspectual advice is invoked, the the system takes over, intercepting the method invocation. The intercepted method invocation is reified as an object, and passed as the sole argument to the wrapping aspectual method. The aspectual method eventually returns a reified return-value object, which is unpacked to reveal the real return value of the intercepted method call. This real value is then returned to the caller, from which point the program continues unchanged.

The system generates two participants from the signature of an **aspectual** method: these are the classes that represent reified method calls (from the type of the aspectual method argument) and return-values (from the return type of the aspectual method). The participant's default API consists of one expected method, on the method-call class: **expected** RetVal `invoke()`, where RetVal is assumed to be the return-value class. The generated participants are completely opaque appart from the `invoke` method, allowing them to be used to represent method calls of any signature,

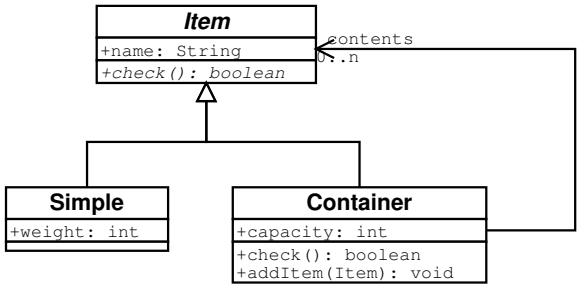

Figure 1: A UML representation of base.

and return values of any type (including raised exceptions or void methods).

To use an aspectual method, it is attached to a host method, allowing it to advise invocations of the host method. Only at the attachment of an aspectual method do we know the signature of the wrapped method, so at that point, attachment-specific implementations of the opaque participants are generated, allowing the attached aspectual method to access the reified method via the opaque interface.

Optionally, the aspectual method can request that the opaque interface be lowered, by constraining the types of (some) arguments or return value. This significantly reduces the genericity of the aspectual method, as it is now restricted to only advise methods which fit the constraints. However, the tradeoff allows the programmer to access method arguments and results, for example re-invoking an idempotent method if it fails on a time-out error, or verifying invariants on methods and return values.

3. CHALLENGE PROBLEM

In order to evaluate the Aspectual Collaboration approach in a more realistic programming example, we implement identical behavior in Aspectual Collaborations, Hyper/J, and AspectJ, and compare the code.

The example is a typical exercise in recursive programming with the composite pattern [16]. A container (composite) may contain items (leaves) or other containers (components). Each container has a maximum capacity, and the task is to verify that no container is over its capacity limit.

The core of the application is written in pure Java, and is common for the three languages. Listings 5 and 6 show the complete code for the base implementation , while a graphical illustration is in Fig 1 . When compiled with `javac` and run, it prints out a complaint that container `c2` is over its weight limit, and adding one banana overloads container `c1` as well.

To detect the capacity violation, base uses straight forward recursion over the containment graph. The common superclass Item declares an abstract method `check` which is implemented in the concrete subclasses Simple and Container to both check the capacity (print out a complaint if above capacity) and also return the total weight of the container and its contained Items. The concrete subclasses differ in that Container has a – possibly empty – vector of contained Items and a maximum capacity, while Simple has a weight but cannot contain any other Items.

Listing 5: Base program, part 1

```
package base;
import java.util.*;
abstract class Item {
  String name;
  public abstract int check();
}
class Container extends Item {
  Vector contents;
  int capacity;
  public static Container make(String n,int c) {
    Container res = new Container();
    res.name = n;
    res.capacity=c;
    res.contents = new Vector();
    return res;
  }
  public void addItem(Item i) {
    contents.add(i);
  }
  public int check() {
    Iterator it=contents.iterator();
    int total = 0;
    while(it.hasNext()) {
      Item child = (Item)it.next();
      total+=child.check();
    }
    System.out.println("Container "+ name +
                       " weighs "+total);
    if(total>capacity){
      System.out.println("Container "+ name +
                         " overloaded");
    }
    return total;
  }
}
```

3.1 Benchmark: A Caching Aspect

The task we set ourselves is implementing a caching aspect for the containers. Inspection of the methods `check` and `addItem` suggests that adding an Item to a Container doesn't invalidate any of the container's contents (subcontainers), but may indirectly invalidate the container and the containers containing the modified one (parent containers). Thus, our programming challenge is to:

1. Add a backlink from an Item to its parent Container. That is, make the `contents` association bi-directional.

2. Every time an item is added to a container, make sure the invariant of each Target pointing to the Source that contains it is maintained [5].

3. Intercept calls to `check`, returning a cached value if one exists, else invoking and caching the result.

4. Every time an item is added to the container, invalidate the cache of the container and all parent containers.

We wish to determine how difficult these tasks are to solve using a reusable style of Aspect-Oriented Programming.

[5]For clarity, we ignore the possibility of a Target being in several Source containers.

Listing 6: Base program, part 2

```
package base;
import java.util.*;
class Simple extends Item {
  int weight;
  public static Simple make(String n,int w) {
    Simple res = new Simple();
    res.name = n;
    res.weight = w;
    return res;
  }
  public int check() {
      System.out.println(
          "Simple object "+name+
          " weighs "+weight);
      return weight;
  }
}
public class Main {
  static public void main(String[] argv)  {
      Container c1= Container.make("Container 1",4);
      Container c2= Container.make("Container 2",1);
      Container c3= Container.make("Container 3",1);
      Simple apple= Simple.make("apple",1);
      Simple pencil= Simple.make("pencil",1);
      Simple orange= Simple.make("orange",1);
      Simple kiwi= Simple.make("kiwi",1);
      Simple banana= Simple.make("banana",1);

      c3.addItem(kiwi);      // c3 weighs 1
      c2.addItem(c3);        // c2 weighs 1
      c2.addItem(apple);     // c2 weighs 2 overload!
      c1.addItem(orange);    // c1 weighs 1
      c1.addItem(pencil);    // c1 weighs 2
      c1.addItem(c2);        // c1 weighs 4
      c1.check();            // c1 is ok
      c2.addItem(banana);    // this overflows c1
      c1.check();            // so check will complain
  }
}
```

Listing 7: Cache a method.

```
1  collab caching;
2  import java.util.*;
3  participant C {
4    ChdRetVal cachedValue;
5    void clearCache() {{
6      System.err.println("clear cache");
7      cachedValue = null;
8    }}
9    expected Vector allInvalidated();
10   aspectual RV invalidate(EM e) {{
11     RV retval = e.invoke();
12     Iterator inv =allInvalidated(). iterator ();
13     while (inv.hasNext())
14       { ((C)inv.next()).clearCache(); }
15     return retval;
16   }}
17   aspectual ChdRetVal cachedmeth(ChdMth e) {{
18     if (cachedValue==null)
19       { cachedValue = e.invoke(); }
20     else
21       { System.err.println("using cached value"); }
22     return cachedValue;
23   }}
24 }
```

4. ASPECTUAL COLLABORATIONS SOLUTION

The Aspectual Collaboration solution consists of two generic and reusable collaborations (Listings 7 and 8), one glue collaboration (Listing 9) which adapts one interface to the other, and finally an attachment of all three to the existing (and unmodified) base behavior (Listing 10). A glue collaboration maximizes the reusability of our solution by collecting the all context-specific code into one collaboration that *does* encode assumptions about the base package and the two other collaborations, allowing the two "main" collaborations (caching and backlink)to remain natural and highly reusable.

Since the AspectJ and Hyper/J solutions are similar, this section also functions as the main discussion of the functioning of the concerns. The other languages' sections will mainly discuss differences between the implementations.

4.1 Caching

Caching a method is implemented by caching in Listing 7. The collaboration has one participant with two aspectual methods: `cachedmeth`, to be wrapped around the method to be cached, and `invalidate`, to wrap the methods which invalidate the cache. The collaboration also has one expected method `allInvalidated`, the implementation of which should return a vector of the objects to be invalidated. From a caching perspective, the design is straight forward.

To use the collaboration, it will be attached to a host collaboration, with a method to be cached and an invalidating method advised by these aspectual methods, and the some implementation of our expected method.

When the wrapped cached method is invoked, the caching collaboration takes over, reifiying the method call as a ChdMth object[6], and passing that to `cachedmeth` to implement caching logic (line 7.17). The caching logic first checks whether we need to invoke the cached method, by comparing the value in instance variable `cachedValue` against `null` (line 7.18). If we need to, we proceed to `invoke` the reified method call (line 7.19), storing its reified result in the instance variable `cachedValue`. Otherwise, we print a self congratulatory message (line 7.21). Finally, we return the reified result.

The `invalidate` method is invoked by the collaboration with the intercepted invalidating method call reified as a EM object . The method immediately (line 7.11) invokes the wrapped method, so `invalidate` implements "after" advice. We then (line 7.13) iterate over all the objects to be invalidated (calculated by whatever implementation of the expected method was provided (line 7.12)), calling `clearCache` on each one (line 7.14). Finally, `invalidate` returns the reified return value from the wrapped method.

The `clearCache` method just sets the `cachedValue` variable to `null` (line 7.7). This is an unambiguous representation of a clear cache (as opposed to a cache containing the value `null`), as reified method calls returning void or `null` will always return a reified result object of type ChdRetVal:

[6]We remind the reader that the participants referenced in the signature of an aspectual method (ChdRetVal and ChdMeth in the case of `cachedmeth`) are automatically generated if not [partially] specified by the programmer. In the next collaboration (backlink), we see why we sometimes want to partially specify some participants.

Listing 8: Add and maintain backlinks.

```
1  collab backlink;
2  import java.util.*;
3  participant Source {
4    expected Vector targets;
5    aspectual RV modifyTargets(ModifyM e) {{
6      int sizebefore = targets.size();
7      RV rv = e.invoke();
8      int sizeafter = targets.size();
9      Target targ = e.t;
10     if (sizebefore<sizeafter)
11       { targ.back = this; }
12     else
13       { targ.back = null; }
14     return rv;
15   }}
16 }
17 participant Target {
18   Source back;
19   Source getSource() {{ return back; }}
20 }
21 participant ModifyM {
22   expected Target t;
23 }
```

reified method calls are guaranteed to always return a reified return value, regardless of whether they return normally or throw an exception.

Notice how the reification API, which encapsulates the details of the wrapped method and intercepted calls to it, allows us to write a caching collaboration which is complete oblivious to the signature of the cached method or its invalidator. Additionally, this illustrates a scenario in which the aspectual method chooses *not* to invoke the wrapped method, instead returning the result from a previous invocation. Both of these are made possible by our reification of the connection between concerns into the domain of the language.[7]

Peeking ahead, we surmise that caching participant C will likely be mapped to Container, with `cachedmeth` advising `check` and `invalidate` advising `addItem`. However, we have not dealt with how `allInvalidated` will be implemented.

The collaboration assumes that there is exactly one method that invalidates the cache, *and* that the method is on the same object as the one cached. The restriction can be alleviated both by programming patterns and language features not introduced in this paper. However, this would have complicated an already large example.

4.2 Backlinks

The second "main" collaboration is backlink, (Listing 8), and corresponds to the programming task of adding and maintaining a backlink from an item to its parent container. It is expressed as three participants. The Source participant expects to be provided with a variable holding a vector of `Target` objects, and to intercept method calls to a method that modifies that vector.

When a modifying method call is intercepted, `modifyTargets` (line 8.5 in Listing 8) takes over, with the intercepted method-call reified as a `ModifyM` object and passed to the method as the only argument `e`. The first thing `modifyTargets` does (line 8.6) is to capture the size of the `targets` vector, before invoking – and capturing the return value of – the wrapped method (line 8.7). `modifyTargets` requires access to the arguments of the intercepted method call (presumably the element added or removed), and we achieve this by adding an **expected** variable `t` to the ModifyM participant[8] (line 8.22), thereby constraining `modifyTargets` to be able to wrap only methods taking [at least] an argument of type Target. We can use the expected field (`e.t`) to access (both to read and modify) that argument (line 8.9). Depending on whether the vector grew or shrank (line 8.10), we either set the backpointer to `this` (line 8.11) or to `null` (line 8.13), maintaining the declared invariant.

Let us briefly peek ahead as to how this collaboration will be related to the host collaboration, base. The result of the aspectual behavior is to maintain an invariant that `Target` objects know which `Source` object points to them. What we want to achieve is to have Items know in which Container they are stored. Thus Source will need to be mapped to Container (from Listing 5), and Target to Item. The method `modifyTargets` will likely be wrapped around `addItem`, and we can guess that the variable `contents` will be provided to `targets`. [9]

4.3 Adapter

Since base.Container contains both the method we wish to cache (`check`) and the method to invalidate the cache (`addItem`), we have already deduced that we are going to need to attach caching.C to base.Container. At this point we would start to write such an attachment specification, but we notice that we don't have a suitable implementation for the cache's `allInvalidated` method: a method that returns *all* parent containers of `this`. We *do* have a method that returns the immediate parent container: backlink's `getSource`. Thus we have the programming task for the adapt collaboration: create a method that returns the transitive closure of `getSource`.

The behavior of adapt in Listing 9 is straight forward: `allContainers` merely calculates the transitive closure of `getContainer`. This collaboration is written in a slightly different style than the others, in that it is written with intent to be used once, in one known context. Hence, we make it as easy as possible to attach, exactly mimicking the inheritance structure of the intended use, and even using the same participant names.[10]

4.4 Attachment

In order give our base access to the caching behavior, we need to attach the collaborations above to the base collab-

[7]It would of course be trivial to provide syntax for the common cases of before and after methods that don't wish to control or inspect the reified method, but would add yet another feature to present, without adding to the power of the language.

[8]The participant ModifyM is only partially specified: the system will recognize it as representing a reified method call (by its position in the signature of an aspectual method) and add all the members that would normally be generated for such a participant.

[9]Notice that the host classes are related by inheritance, while the collaboration classes are not. This will not pose a problem for attaching backlink to base, but will influence the requirements for the adapt collaboration.

[10]The correspondence in names is for clarity only. All attachment is explicit, and no the behavior is exactly the same as had we chosen more arbitrary names. It also lets us exercise class disambiguation in the attachment.

Listing 9: Generating a transitive closure.

```
1  collab adapt;
2  import java.util.Vector;
3  participant Item {
4    expected Container getContainer();
5  }
6  participant Container extends Item {
7    Vector allContainers() {{
8      Vector v = new Vector();
9      Container c = this;
10     while (c != null) {
11       v.add(c);
12       c = c.getContainer();
13     }
14     return v;
15   }}
16 }
```

Listing 10: Attaching our collaborations.

```
1  collab cachedbase;
2  extends base;
3  attach backlink, caching, adapt {
4    Item += Target, adapt.Item {
5      provide getContainer with getSource;
6    }
7    Container += Source, C, adapt.Container {
8      provide allInvalidated with allContainers;
9      provide targets with contents;
10     around void addItem(..,Item t,..) do modifyTargets;
11     around addItem do invalidate;
12     around check do cachedmeth;
13   }
14 }
```

oration. Listing 10 illustrates this, generating a new collaboration with the name cachedbase. [11] The attachment sets up pointwise mappings between the participants of the constituent collaborations (caching, backlink, and adapt), and the output participants imported from base (Item, Simple, Container, and Main). Notice that we only mention the participants we are interested in, as the semantics of **extends** has implicitly exported all of base. If we desired to, we could additionally hide some of those participants and members at this point. Likewise, the member mappings we have alluded to in the peek-ahead paragraphs are set up; for example, expected field `targets` is provided with field `contents`.

The attachment processes each participant separately: To create the output participant cachedbase.Item, we start with base.Item (implicit from **extends** line 10.2), and insert the constituent participants backlink.Target and adapt.Item (line 10.4) into it. We don't need to fully qualify Target, as it is the only such participant in the the output or constituent collaborations, but there are two Item participants (from adapt and cachedbase), so we must disambiguate. Once created, the output participant Item's members are linked, providing the expected method `getContainer` with the implemented `getSource`. Again, since these member names are unambiguous, we don't need to specify from which constituent participant each came.

Container is similar, but there are pertinent observations concerning attachment of aspectual methods. We set up `cachedmeth` to intercept the invocation of `check` (line 10.12), at which behavior proceeds as explained for caching. Similarly for `addItem`.

It is not obvious which wrapper of `addItem` (line 10.10 or line 10.11) is executed first. One answer is that it should not matter, as each collaboration should be somewhat semantically complete, and two collaborations whose implementations are so intertwined so to make a difference which is invoked first should really be composed to one more cohesive collaboration rather than added separately. A slightly more satisfying answer is that `invalidate` will be invoked first, as **around** wrappers are processed in program order, and each processing stage builds on the previous method implementation. One can view the method and wrappers like a *Matrioshka* doll, where each doll has control over whether the dolls inside itself are executed. The original `addItem` is the innermost doll, which is reached last – if at all. The outermost doll represents `invalidate`, which has control over the invocation of `modifyTargets`.

Lastly, the correspondence of the argument name `Item t` in the partially specified signature of `addItem` (line 10.10 in 10) to the expected variable in backlink.ModifyM (in Listing 8) *is* important. This specifies that the first argument of type Item in method `addItem` is to be exposed to the backlink aspect, as the expected field `t`.

5. HYPER/J SOLUTION

Hyper/J is designed for capturing and manipulating slices of concerns. HyperSlices are extracted from compiled applications, and composed into running applications. The Hyper/J solution for our scenario is structurally very similar to that of Aspectual Collaborations, but we will find that small differences in semantics give the result a distinctly different flavor. An interesting property of Hyper/J is that all examples are pure Java, with *all* concern related information in the HyperModule file. This both hampers and helps readability. Each concern becomes very easy to understand, but we must look elsewhere to understand even the rudiments of how the concerns [can] fit together.

5.1 Caching

Caching relies on "around" advice, which Hyper/J does not have, so we must simulate this behavior.

There are two ways to achieve the required "around" behavior in Hyper/J: splitting the "around" into "before" and "after" advice, or manually mapping the advised method into the HyperSlice, so that it can be invoked explicitly.

Manual mapping gives the concern much more flexibility as to how to affect the wrapped method, but hampers reuse by requiring us to know the exact signature of the wrapped method. Caching (Listing 11) differs markedly from the Aspectual Collaboration version, as it illustrates manually mapping the intercepted method call into the concern.

Hyper/J is able to provide access to the result of method invocation without needing to capture the bracketed method explicitly, but as we additionally need to be able to control *whether* the bracketed method is invoked at all, we must make it explicit in the concern. This in turn implies that we must hard-wire the exact signature of the method we are caching. Peeking ahead, we intend that the original `check` method on base.Container be hidden somehow and composed with `oldcachedmeth` (line 11.16), and `newcachedmeth` (line

[11] As all required interfaces have been fulfilled, this collaboration is simultaneously a normal Java package.

Listing 11: Cache a method.

```
1  package caching;
2  import java.util.*;
3  abstract class C {
4    boolean cacheValid = false;
5    boolean cachedValue;
6    abstract Vector allInvalidated();
7    void clearCache() {
8      System.err.println("clear cache");
9      cacheValid = false;
10   }
11   void invalidate() {
12     Iterator inv = allInvalidated().iterator();
13     while (inv.hasNext())
14       { ((C)inv.next()).clearCache(); }
15   }
16   abstract boolean oldcachedmeth();
17   boolean newcachedmeth() {
18     if (!cacheValid) {
19       cachedValue = oldcachedmeth();
20       cacheValid = true;
21     }
22     else { System.err.println("using cached value"); }
23     return cachedValue;
24   }
25 }
```

Listing 12: Add and maintain backlinks.

```
1  package backlink;
2  import java.util.*;
3  class Source {
4    Vector targets;  // deferred
5    int sizebefore;
6    void modifyTargetsBefore() {
7      sizebefore = targets.size();
8    }
9    void modifyTargetsAfter(Object[] margs) {
10     Target targ = (Target) margs[0];
11     int sizeafter = targets.size();
12     if (sizebefore<sizeafter)
13       { targ.back = this; }
14     else
15       { targ.back = null; }
16   }
17 }
18 class Target {
19   Source back;
20   Source getSource() { return back; }
21 }
```

11.17) to become the one visible as `check`. This allows the caching behavior to invoke the original behavior (line 11.19) when necessary. Setting this up is possible, but somewhat fiddly, and requires features that are not available in the current release of Hyper/J. However, the developers have kindly furnished us with a pre-release version that is able to perform this mapping.

Notable also is that we have chosen to use the **abstract** to indicate that the `allInvalidated` method (line 11.6) is required. Unlike fields, where comments are the only option, methods can be annotated as required either by declaring them abstract, or else implemented to throw the Hyper/J-specific UnimplementedError exception. This sentinel exception is recognized by the composition system to indicate a deferred method, and thus the stub method body is omitted when two methods are composed. The benefit of using the sentinel approach is that the class remains instantiatable, while the abstract method approach has the benefit that it is statically obvious that some method in the class is deferred. The latter approach suffers from the additional drawback of forcing each subclass of the abstract class to be abstract as well, even if they have no abstract methods, as they inherit the deferred method, which they cannot override, as that would hide the behavior the deferred member is supposed to receive through composition.

5.2 Backlinks

Listing 12 shows the Hyper/J implementation of the backlink concern. The most striking difference from the Aspectual Collaboration version is that `modifyTargets` is here split into two methods. We simulate "around" advice by splitting it into "before" and "after" advice to a method. Splitting allows the concern to remain oblivious to the signature of the wrapped method, but is only applicable when the inner method is to be called exactly once, the result is not handled, and we are certain that the splitting is thread-safe (more on this in the next paragraph).

The before method `modifyTargetsBefore` just stores (line 12.7) the size of the vector in an instance variable, so that it is available in `modifyTargetsAfter` method. This adds a small but non-zero risk of race-conditions in multi-threaded code, as a second thread could overwrite this variable before `modifyTargetsAfter` has read it (on line 12.12). The most straight-forward way to protect against this would be to use the current thread object as a key into a hashtable where per-thread instance variables are stored. Additionally, we ought to protect against recursive invocation by keeping a stack of instance variables.

The signature of the method advised by this simulated "around" is constrained in solution as well, but in this case it is constrained by a cast in the after method body (line 12.10), trading off static safety against some runtime flexibility. We open ourselves up to a runtime error by not checking the cast first, but as this is a simple example and we know how this code is going to be composed, we feel this is excused.

Unfortunately, we are reduced to comments to indicate that `targets` is not implemented in this concern, as the techniques Hyper/J uses to identify deferred members work only for methods.

Peeking ahead, we foresee that two methods `modifyTargetsBefore` and `modifyTargetsAfter` being composed as **before** and **after** brackets to `addItem`. The `margs` argument (line 12.9) will be constructed by the HyperModule to pass in the arguments from the bracketed method.

5.3 Adapter

As in the Aspectual Collaboration solution, there is a mismatch between the interfaces of the backlink and caching packages. Listing 13 illustrates how these can be resolved in exactly the same manner as for Aspectual Collaborations. Notice however, that the deferred method `getContainer` is implemented to throw UnimplementedError (line 13.6) rather than being abstract. Had it been abstract, then class Item would also have been abstract (line 13.4), which in turn would have forced `Container` (line 13.9) to be abstract as well.

Listing 13: Adapting the Concerns.

```
1  package adapt;
2  import com.ibm.hyperj.UnimplementedError;
3  import java.util.Vector;
4  class Item {
5    Container getContainer() {
6      throw new UnimplementedError();
7    }
8  }
9  class Container extends Item {
10   Vector allContainers() {
11     Vector v = new Vector();
12     Container c = this;
13     while (c != null) {
14       v.add(c);
15       c = c.getContainer();
16     }
17     return v;
18   }
19 }
```

5.4 HyperSlice composition

Listing 14 shows the Hyper/J specification for identifying and composing the HyperSlices we presented above. The specification consists of three parts that optionally can go into separate files. The **hyperspace** specification (lines 14.2-10) identifies which classes are participating the the composition. These classes are partitioned into hyperslices by the **concerns** specification (lines 14.11-18). Finally, the **hypermodule** (lines 14.19-45) chooses which of these hyperslices to compose, and how their contents relate.

The **relationships** clause has indentation for easier reading, but is actually a sequence of flat declarations. The **mergeByName** declaration is used mainly at the class level in this example, automatically composing the similarly named classes (for example adapt.Container and base.Container). The result of the composition of hyperslices is the union of all their classes, minus the classes that have been composed, either through explicit annotations or implicit by-name merging. The situation at the level of class members is analogous.

There are two side effects of the way Hyper/J composes modules. Most importantly, it implies that hypermodules lack encapsulation, as there is no way to avoid exporting every member and class in a slice. It *is* possible to rename members and classes, and implement a naming convention to indicate which module contents should not be accessed from the outside world. However, this does not stop accidental name matches from invoking the **mergeByName** rule, with unintended effects.

Secondly, lack of encapsulation makes it difficult to predict the interface of a hypermodule without running the Hyper/J tool. A HyperSlice can be defined as a slice of a composed hypermodule, which in turn could be composed from slices, creating an import chain of arbitrarily length. The lack of encapsulation means that we must start at the sources and mentally propagate through the whole chain to build up the final interface, rather than having it declared as part of the hypermodule. The two effects also interact, in that adding a method to a module early in the chain can lead to a spurious match later on, with unintended effects. The situation is similar to that of accidental inheritance [31] or accidental method capture [32].

Listing 14: Attach the slices.

```
1  −hyperspace
2    hyperspace H
3    composable class backlink.Source;
4    composable class backlink.Target;
5    composable class caching.C;
6    composable class adapt.Item;
7    composable class adapt.Container;
8    composable class base.Item;
9    composable class base.Container;
10   composable class base.Simple;
11 −concerns
12   class backlink.Source: Feature.Back
13   class backlink.Target: Feature.Back
14   class caching.C:       Feature.Cache
15   class adapt.Item:      Feature.Adapt
16   class adapt.Container: Feature.Adapt
17   class base.Item:       Feature.Base
18   class base.Container:  Feature.Base
19 −hypermodules
20   hypermodule CachedComputation
21   hyperslices: Feature.Base, Feature.Cache,
22                Feature.Adapt, Feature.Back;
23   relationships:
24     mergeByName;
25     compose class Feature.Back.Target
26       with additionally class CachedComputation.Item;
27     equate operation Feature.Adapt.getContainer,
28                      Feature.Back.getSource;
29     compose class Feature.Back.Source, Feature.Cache.C
30       with additionally class CachedComputation.Container;
31     equate operation Feature.Cache.allInvalidated,
32                      Feature.Adapt.allContainers;
33     equate variable Feature.Base.Container.contents,
34                     Feature.Back.Source.targets;
35     forward operation CachedComputation.Container.check
36       to operation Feature.Cache.newcachedmeth;
37     equate operation Feature.Cache.oldcachedmeth
38       to operation Feature.Base.check;
39     bracket "Feature.Base.Container"."addItem"
40       before Feature.Back.Source.modifyTargetsBefore()
41       after  Feature.Back.Source.modifyTargetsAfter
42                                    ($ArgumentArray)
43       bracket "Feature.Base.Container"."addItem"
44         after Feature.Cache.C.invalidate()
45 end hypermodule;
```

6. ASPECTJ SOLUTION

Clarke and Walker's [10] translation of composition patterns to AspectJ is applicable to Aspectual Collaborations. We follow their transformation in spirit, if not to the letter, generating a surprisingly idiomatic AspectJ program. In our translation, we have try retain the focus on the reusability inherited from the original Aspectual Collaboration version to keep the comparison between apples and apples.

6.1 Caching Aspect

Listing 15 shows the AspectJ implementation of Caching. It is similar in behavior to Aspectual Collaborations' solution, so we only cover the language specific details.

We delay discussion of the use of interfaces to model participants until the Back aspect, as the discussion is more concrete in that context.

The `cachedmeth` around advice illustrates the unconventional genericity mechanism of AspectJ's around advice [4]. The return type Object (line 15.12) indicates in AspectJ's semantics that this method can advise methods of *any* sig-

Listing 15: The Caching aspect.

```
1  abstract aspect Caching {
2    interface C {
3      java.util.Vector allInvalidated();
4    }
5    Object C.cachedValue;
6    void C.clearCache() {
7      System.err.println("clear cache");
8      cachedValue = null;
9    }
10   abstract pointcut cachedmeth(C t);
11   abstract pointcut invalidate(C t);
12   Object around(C t): cachedmeth(t) {
13     if(t.cacheValue==null) {
14       t.cachedValue = proceed(t);
15     } else { System.err.println("using cached value"); }
16     return t.cachedValue;
17   }
18   before(C t):invalidate(t) {
19     java.util.Iterator it = t.allInvalidated().iterator();
20     while (it.hasNext())
21       ((C)it.next()).clearCache();
22   }
23 }
```

Listing 16: The Back aspect.

```
1  import java.util.*;
2  abstract aspect Back {
3    interface Source { Vector getTargets(); }
4    interface Target { }
5    abstract pointcut modifyTargets(Source s, Target t);
6    void around(Source s, Target t): modifyTargets(s,t) {
7      Vector targets = s.getTargets();
8      int sizebefore = targets.size();
9      proceed(s,t);
10     int sizeafter = targets.size();
11     if (sizebefore<sizeafter) t.back = s;
12     else t.back = null;
13   }
14
15   Source Target.back;
16   Source Target.getSource() { return back; }
17
18
19 }
```

nature. If the advised method returns a primitive or void value, it will be wrapped in the proper java.lang wrapper class before being returned from `proceed` (line 15.14). The returned object from the advice will be downcast (and possibly extracted) to the type indicated by the signature of the advised method. Although the AspectJ compiler protects the programmer against so called stupid casts [20] during this procedure, any use of Object (as when retrieving return values from a Collection) can easily lead to casting errors in generated code not visible to the user. Understanding why this occurred requires the programmer to be a language "wizard".

For simplicity, we assume that a cached method never returns `null`, and use this as a sentinel for cache validity. An additional boolean variable, as in the Hyper/J example, could have been used to keep track of the validity of the cached value.

6.2 Back Aspect

Listing 16 shows the AspectJ implementation of Back. We write an abstract aspect which is made concrete in a subclasses by providing the necessary application-specific information. AspectJ uses interfaces to declare types, which are then populated with behavior by introductions and **implemented** by the base program, while the two other approaches' concerns declare their behavior in classes which are externally composed with the base program.

The situation in AspectJ is analogous to that of Java: it is conscidered good programming style to program against an interface, but up to the programmer to decide when and where to do so. The AspectJ approach shares with Java the problem that one cannot instantiate an interface directly, but rather need an abstract factory pattern. We discuss the advantages and disadvantages of interfaces pertaining to attachment in the section on concrete aspects (section 6.3).

The interfaces describe the aspect's required interface. Due to restrictions in interfaces, the required interface can only contain methods. This can be worked around by instead adding getters and setters to the interface, and introducing methods to fulfill this interface . Thus, in AspectJ the situation with fields is even more restrictive than in Hyper/J, where we were forced to use comments to indicate a deferred field, but were able to compose them in the hypermodule.

Provided methods are added to host classes directly via AspectJ's introduction mechanism, which interacts gracefully with interfaces, adding the introductions to the implementing class rather than the interface (which of course could not be an interface were it to contain code). Specifying an interface and its added behavior separately both separates provided from deferred behavior textually (deferred methods are directly on their interfaces, while provided behavior is in the aspect body), and also tangles the provided behavior for all the participants into one class body.

Advice is declared against abstract pointcuts. AspectJ has a mature join point model, and we are easily able to express our intended behavior. The format of the advice method differs from the Aspectual Collaboration and Hyper/J approach in that we explicitly pass in the receiver of `modifyTargets`, while `this` refers to the object representing the reified aspect. This allows advice to be quite general, wrapping methods of varying signatures in different classes.

6.3 Concrete Aspects

Listing 17 illustrates how the abstract aspects can be instantiated for the current application through subclassing. We attach the participant interfaces of the two abstract aspects to classes by having the class implement the interface. This is perhaps the biggest difference in approach between AspectJ, and Hyper/J or AspectJs, in that the latter two specify composition externally, while the former does so internally to the application.

We have moved the adapt concern into the connection aspect, as it is tightly bound to both of the concerns it adapts. AspectJ provides a very natural way to specify this sort of adaptation code.

The implicit *effect* of supplying a concrete point-cut to the abstract aspect is that its advice becomes attached to the application at the join points specified. However, using interfaces has explicit effects on the program as well: it becomes

Listing 17: Concrete instances of the Aspects.

```
1  import java.util.Vector;
2  aspect Concrete1 extends Back {
3     declare parents: Item implements Target;
4     declare parents: Container implements Source;
5     pointcut modifyTargets(Source s,Target t):
6       target(s)
7         && call(* Container.addItem(Item))
8         && args(t) ;
9  }
10 aspect Concrete2 extends Caching {
11    declare parents: Container implements C;
12    pointcut cachedmeth(C t):
13      target(t) && call(* Item.check(..));
14    pointcut invalidate(C t):
15      target(t) && call(* Container.addItem(..));
16 }
17 aspect Adapt {
18   public Vector Container.allInvalidated() {
19     Vector v = new Vector();
20     Container c = this;
21     while (c != null) {
22       v.add(c);
23       c = (Container) c.getSource();
24     }
25     return v;
26   }
27   public Vector Container.getTargets() {
28     return contents;
29   }
30 }
```

visible outside the package that a class has implemented a participant interface. For example, `anObject instanceof Target` tests whether the back concern has been attached to `anObject`. Ironically, while it *is* apparent at runtime that Item implements Target, this is not apparent statically by inspecting the code, unless we come across Concrete1. A naming convention can easily ameliorate this, however.

The objection to runtime composition tests may appear more of an aesthetic than Software Engineering issue, but the mechanism of attaching participants by implementing interfaces has consequences for type safety as well. If the back collaboration were attached twice in an application, there would be two classes that now have the type Target. Since we intend to reuse back, there may be additional such classes in library code. The problem is that the behavior provided to the application by the concern is written against the interface types, while the methods implementing the concern's required interface will likely be casting these type to the implementation classes[12]. Ie, it is to pass *any* Target class to a method with that argument type, but that method's implementation may likely assume that the Source returned by `getSource` is actually a Container. This assumption would be foolish in normal code implementations of an interface, but since a concern is declared as a unit, it makes sense for the programmer to assume that the attachment is also a unit.

Lastly, we were lucky that back and cache didn't have any name clashes. While AspectJ offers aspect-local methods for the concern's implementation behavior, the expected methods on the interfaces *must* be implemented by public methods with exactly that name.

[12]Our examples do not involve any expected methods that take participant types as arguments, so this case does not appear.

7. DISCUSSION AND LESSONS

The previous sections offered a detailed comparison between Aspectual Collaborations, Hyper/J, and AspectJ. In this section we summarize the lessons learned from the comparison.

7.1 Hyper/J

Hyper/J and the work on Multi-dimensional Separation of Concerns [39, 40] generalizes the ideas behind Subject-Oriented Programming [17, 35] by moving to finer grained units of combination. A HyperSlice is a named set of classes containing sets of methods and fields. The slice can be added to new classes in a very similar way to collaborations.

Hyper/J takes a stricter approach to phase distinction than Aspectual Collaborations, performing all concern related operations on pure Java programs, with absolutely no language modifications. In contrast Aspectual Collaborations add some additional syntax to Java, requiring them to be compiled as well as composed by `acc`.

HyperSlices were somewhat inflexible to use for our challenge problem, requiring us to hardwire the signature for an around method, and requiring downcasts of intercepted method arguments in before and after advice. HyperSlices do not control the visibility of members and classes, opening slice composition up to inadvertent name capture [31]. However, Hyper/J does well at type-safe reuse, achieving reuse without requiring the hosts to share types and hence make casting errors more likely.

Hyper/J additionally offers several features that did not come up in the example. Post-hoc remodularisation allows a HyperSlice to be teased out of a set of classes (possibly generated by composing slices, or by compilation of `.java` files) and used separately. Hyper/J also provides several dimensions of composition, of which our examples only used the feature dimension.

7.2 AspectJ

AspectJ from Xerox PARC [41] resulted from the initiative to factor out commonalities in several domain specific aspect languages. Crista Lopes' thesis [27] investigated two of those languages in detail: COOL [29] for specifying the synchronization aspects, and RIDL [28] for specifying data transfer aspects.

AspectJ integrates aspect features tightly into the language, forgoing the semantic and syntactic overhead of module systems, but also the benefits of encapsulation and separate compilation. While this gives rise to very natural specification of aspectual behavior, it comes at a cost of program comprehension, as the lack of encapsulation boundaries for advice forces the programmer to read the whole program to determine whether a join-point has advice. The AspectJ team developed tools to perform such system-wide program comprehension, and integrate join-point feed-back into several IDEs. Analogously, there is no way to protect a join-point against being advised.

AspectJ aspects are reusable by programming abstract aspects against interfaces that are attached at a later time to

the host program.[13] Aspectual advice achieves surprisingly good reuse, both by mentioning only the types necessary in a point-cut signature, and also by the somewhat unorthodox genericity mechanisms of around methods (see section 6.1), which appear to work very well in practice.

Unfortunately, programming abstract aspects against interfaces suffers from low levels of type safety: generating casts in programmer-invisible code that can fail, and forcing multiple uses of an aspect to share types (recall section 6.3), which opens up the program to further casting errors. The use of interfaces interferes with reuse, as it restricts expected members to be methods, and requires name equivalence between expected and provided methods.

7.3 Aspectual Collaboration critique

Aspectual Collaborations are not without flaws: the explicit use of reified aspectual methods may impose a performance penalty, and requiring the user to capture and return a reified result introduces some awkwardness into aspectual methods. However, we gain a very clear insight into the exact behavior of our program, in addition to strong type-safety guarantees and flexible reuse.

Encapsulation allows us to also understand the program in a compositional manner: by looking only at one attachment specification, we can make accurate predictions about how the mentioned parts of the program can communicate with each other. In contrast, AspectJ has long recognized the "come-from" [7] nature of advice [37], and integrate tool support into several IDEs to assist programmer overview of the program. Similarly, while Hyper/J's HyperModule definitions make it explicit *how* concerns are composed, alleviating the "come-from" nature of aspects, the seeming lack of encapsulation makes understanding *what* concerns are composed impossible without performing a full trace of the concern composition history to gather up the accumulated interfaces for the slices.

Adaptive Plug&Play (AP&P) components [34] and the follow-on report [25] are the immediate precursors to Aspectual Collaborations. AP&P components are rooted in Holland's executable contracts [19] and in Rondo [33]. This work builds on [25], but with significant modifications from experience with implementation, and with a very different attachment / matching model stemming from clarified semantics. The difference in attachment and matching reflects that AP&P components are aimed at being language level components rather than system structuring modules. Additionally, they AP&P components offered somewhat weaker aspectual capabilities.

Mezini and Herrmann [18] discuss a software engineering environment capable of combining dynamic plugability, separate compilation, and aspectual attachment. It is unclear how their PIROL system deals with type safety.

8. RELATED AND FUTURE WORK

In this section we bring in work related to Aspectual Collaboration and future research issues.

8.1 Module Systems

Module systems are effectively the dual of AspectJ, offering powerful encapsulation and reuse support, but no support for aspectual or concern-oriented features. However, just as AspectJ can still achieve reuse without encapsulation, it is possible to achieve some aspectual features without tool support – as in Hyper/J's implementation of an around method. Indeed, we are currently investigating the possibility to use a third-party module system as a back end, rather than our own solution, in order to free up developer resources.

Jiazzi [30] is the implementation of Units [15] for Java. Jiazzi reuses Java's core composition feature – inheritance – and the Open Class pattern to construct the resulting classes from partial implementations. Late binding is used to allow mutually recursive dependencies between modules. A Jiazzi implementation of the caching example would likely look very similar to the Hyper/J version, but instead of capturing the original method explicitly and then relinking the resulting class to swap in the cached version instead of the original, In Jiazzi we would specify that we expected the cached method to be declared on a superclass, and then proceed to override it, calling the original method with a super call.

It is interesting to note that although developed completely separately, the current back end for Aspectual Collaborations and Jiazzi are strikingly similar. The main differences are how the finished classes are assembled (Jiazzi favors inheritance, while we manually combine and link participant `.class` files) and the fact that we favor intrinsic typing for collaborations, while Jiazzi allows reuse of a unit type for several unit implementations.

Mixin-Layers [38] represent a collaboration as a layer of mixin classes. This layer is treated as a unit, so that the interface to the mixin layer is a set of superclass imports, and a set of class exports. Mixin layers can be composed creating composite layers, allowing modular construction of complex programs. Both Jiazzi and Aspectual Collaborations generalize layers, in that both can represent layers as a programming pattern, but can also represent other forms of modular construction.

8.2 Component Systems

Component Systems can be seen as a restricted form of module system, where imports are not classes, but rather objects and operations on objects. The distinction is that component systems seldomly offer any form of parametricity over types, relying rather on subtype polymorphism. Furthermore, the result of linking components typically does not generate a new type, but rather links existing component instances together to construct complex behaviors from simpler ones.

ArchJava [3, 2] is a a modern example of a component system. Components communicate with each other over named sets of methods, called ports. Component instances can be connected both statically or dynamically, and in both cases, the system guarantees that components only communicate to their neighbors, ensuring *communication-integrity*. However, the authors point out that this applies only to method invocations – shared object references can still be propagated through the system allowing communication to pass via the shared object.

Aspectual Collaborations are not able to conveniently express, nor guarantee communication integrity for dynamic component connection, but are able to quite well for the static case. Using encapsulation, we are able to make state-

[13] It is unclear whether an abstract aspect can be separately compiled, or whether it is recompiled with the application.

ments not only which components a component is able to talk *to*, but more strongly which components it can talk *about*. If a component doesn't import another component's type, direct communication between them is impossible. This applies additionally to auxiliary classes and objects that are passed between components. If a component has only a limited view of a class (for example omitting a sensitive field), then we can statically guarantee that this field cannot be directly manipulated by the component. If a component does not know about a class at all, it cannot communicate via objects of that type at all.

Interestingly, analysis of the ArchJava papers has yielded a programming pattern – tentatively called the *component/-port* pattern, which significantly enhances the readability of an Aspectual Collaboration by separating out connections between collaborations onto port classes.

8.3 Modeling Languages

Moving away from implementation to the modeling arena, we see some connections to modeling efforts.

Composition Patterns [9] adds the concept of composition patterns to UML [6]. The implementation suggestions for composition patterns [8] strongly influenced the implementation of reusable aspects and hyperslices in this paper, and we propose that Aspectual Collaborations do an particularly elegant job of representing Composition Patterns. It is unclear whether Composition Patterns capture multiple attachments of a collaboration, and how sharing of members between such attachments would be expressed.

Catalysis methodology [12] has a strong emphasis on modeling collaborations. Catalysis uses a *common model of attributes*. In comparison, we use a participant graph, and have built-in support to express aspectual decompositions.

8.4 Future Work

There are several subtle implementation issues that need to be dealt with in future work. These are issues that we have solved at the language design level, but not yet implemented.

Parameterization. By adding types to collaborations, it becomes feasible to express attachments which are abstracted over the exact collaboration attached. By passing collaboration-valued parameters as arguments to attached-collaborations, very complicated behaviors can be succinctly expressed. The challenge is to develop a type system that allows flexible use while capturing errors early. Errors will always be caught at compilation time, but it is desirable to catch them when the parameterized collaboration is defined, rather than used.

Point-cuts in the interface. The interface to an Aspectual Collaboration currently contains only participants and members. Thus, in order to add advice to a member, it must be in the interface, and hence visible. It would be nice to be able to decouple these concepts, also putting point-cuts into the interface of an Aspectual Collaboration. Thus, the collaboration could export the ability to add advice to a member, without exporting the member itself.

Constructors. The main problem with constructors is that they are the only methods which should be merged, rather than kept separate. When two constituent participants are mapped to the same output participant, we want creation of an output-participant object to invoke all three (two constituent, and one output) constructors. Each participant may actually contribute several constructors, which in turn may lead to several inheritance chains of constructors. Since a constructor may invoke any method of a class, we have to be *very* careful about the order in which class initialization methods are invoked. Our current solution is to have a sensible heuristic, and ask the user to specify in the situations where that does not apply. We note that Jiazzi [30] has a similar restriction, requiring all constructors in an inheritance chain to have exactly the same signature.

Change the Back End. Currently, `acc`, the Aspectual Collaboration compiler, compiles a collaboration to plain Java, compiles that, and then works on the generated byte-code for the attachment and composition operations. However, the mechanics of the byte-code manipulation are tedious, and is not the contribution of our research. Instead it may be possible to offload this development burden to a back-end based on Jiazzi, which would allow us to focus on developing the module interface language and aspectual features.

8.5 Possible Extensions

In addition to those implementation issues, the following are natural extensions to this work:

Object Graph Constraints. A key concept of collaborations is that each has its own class-graph, which are fused when one is attached to another. The behavior of a class-graph will in general instantiate classes of that class graph and store the objects in variables. In effect, each collaboration will build its own object-graph. In addition to *building* an object graph, the collaboration also makes assumptions about it–these assumptions are encoded in the code of the collaboration, and take the form of invariants over the object-graph.

Examples of invariants are that a non-zero value for one variable indicates that another is ready to be read, or that two variables of the same type in fact *alias* the same object. The key insight here is that the fused collaborations must make compatible assumptions about their object-graphs, as in addition to sharing a fused class-graph after attachment, they will at runtime also share an object-graph.

It would be helpful to capture these constraints in the interface of the collaboration, so as to be able to catch such attachment errors at compile-time. This can be seen as a special case of contract checking, where perhaps machine analysis can help derive the object-graph (run-time) constraints to be checked at compile-time.

Macro System. We purposefully keep our core language minimal to achieve a simple semantics. However, this leads to programming patterns occurring over and over. Rather than adding these patterns to our core language, one approach would be to desugar a rich input language (perhaps containing "before" and "after" advice) into our core. This desugaring could be written as pre-processing task, but this would force the developers to update it at the whims of the users of the language. A better solution would be to include a macro language, to allow the users to encode their particular programming patterns.

Garbage-collect Participants Because collaborations are closed to future extension apart from the exported interface, we can statically determine all reachable participants and members. Thus, we can safely remove all non-reachable

participants and members, retaining only the exported interface and any non-visible but reachable members. This smaller collaboration becomes very similar to a teased-out HyperSlice.

9. CONCLUSION

A known AOP problem addressed by several authors is the difficultly to tease out and reuse aspects which are tightly integrated into host code [14]. The reason why aspects are often so tightly integrated with the host code is the lack of an encapsulating interface between the aspectual unit and the rest of the system. This paper addresses this issue by combining the power of aspects with the encapsulation power of modules. We demonstrate that writing aspects against formal participant graphs, and attaching them to other participant graphs, helps in making the aspects both more abstract and reusable.

We have presented Aspectual Collaborations, which combine the static properties of modules: encapsulation, external composition, and separate compilation, with the flexible programming power of aspects. Aspectual Collaborations are a wrapper around Java, which adds a module system and support for aspectual behavior and separate compilation. We have shown how the system implements separate compilation of aspectual and additive behavior; allows composition and parameterization of collaborations; and can transparently interface with existing Java programs. We have elided the description of the implementation, which is implemented as a somewhat involved desugaring of aspectual features to a module language back-end.

This paper has compared Aspectual Collaborations against two popular aspect-oriented systems, on a small but reasonable example, to evaluate the overhead of each system, and if possible to gauge whether the presumed overhead of our system is "worthwhile".

We expected Aspectual Collaborations to, by design, be better than AspectJ at reuse, but under perform on a small sized example, since the extra syntax of a module language may be comparatively cumbersome for a small program. Much to our delight, we found that compared to reusable aspects in AspectJ, Aspectual Collaborations are only somewhat more verbose, but at the considerable benefit of separate compilation and type safety.

We expected Aspectual Collaborations to offer modular power similar to Hyper/J, but with better aspectual features. This was borne out. We did find that Hyper/J brackets allow before and after advice to be written fairly easily, but simulating around advice was quite tricky, composing hyperslices with differing names was quite verbose, and controlling the details of the generated output was fiddly. Hyper/J's features work well when working with a set of hyperslices with common and recurring names, allowing its composition functions can be used with the mergeByName matching.

The Aspectual Collaborations advantage shows up when composing very different collaborations, with differing names and participant graph shapes. Additionally, Aspectual Collaborations will do even better when precise control is needed over which members are to be visible from a composed collaboration, and when one collaboration is to be reused several times in different contexts, with the same advice applied to signatures of different types.

10. REFERENCES

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, NY, 1996.

[2] J. Aldrich and C. Chambers. Architectural reasoning in archjava. In *European Conference on Object-Oriented Programming*, 2002.

[3] J. Aldrich, C. Chambers, and D. Notkin. Archjava: Connecting software architecture to implementation. In *International Conference on Software Engineering*, 2002.

[4] AspectJ Team. *AspectJ Programming Guide*. http://aspectj.org/doc/dist/progguide/apbs03.html.

[5] X. P. AspectJ Team. AspectJ home page. http://aspectj.org. Continuously updated.

[6] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison Wesley, 1999. ISBN 0-201-57168-4.

[7] R. L. Clark. A linguistic contribution to goto-less programming. *Communications of the ACM*, 1974. Originally published in Datamation, 1973.

[8] S. Clarke. Designing reusable patterns of cross-cutting behavior with composition patterns. In the Second Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000), 2000.

[9] S. Clarke and R. Walker. Composition patterns: An approach to designing reusable aspects. In *International Conference on Software Engineering*. ACM Press, 2001.

[10] S. Clarke and R. Walker. Separating Crosscutting Concerns Across the Lifecycle: From Composition Patterns to AspectJ and Hyper/J. Technical Report UBC-CS-2001-05, University of British Columbia, Vancouver, CA, 2001.

[11] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N. J., 1976.

[12] D. D'Souza and A. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.

[13] T. Elrad, R. Filman, and A. Bader. Aspect-Oriented Programming. *Communications of the ACM*, 44(10):28–97, 2001.

[14] E. Ernst. Separation of Concerns and Then What? In *Workshop on Advanced Separation of Concerns, ECOOP*, Cannes, France, 2000. Electronic form: *http://www.cs.auc.dk/ eernst/*.

[15] M. Flatt and M. Felleisen. Units: Cool modules for hot languages. In *ACM Conference on Programming Language Design and Implementation*, pages 236–248, 1998.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*, number NU-CCS-93-1, pages 406–431, December 1993.

[17] W. Harrison and H. Ossher. Subject-oriented programming (A critique of pure objects). In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, pages 411–428, Oct. 1993. Published as Proceedings OOPSLA '93, ACM SIGPLAN Notices, volume 28, number 10.

[18] S. Herrmann and M. Mezini. PIROL: a case study for multidimensional separation of concerns in software

engineering environments. In *OOPSLA*, pages 188–207, 2000.

[19] I. M. Holland. *The Design and Representation of Object-Oriented Components*. PhD thesis, Northeastern University, 1993.

[20] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* in *Special Issue of SIGPLAN Notices*, pages 132–146, 1999.

[21] S. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356, April 1993.

[22] G. Kiczales. Beyond the black box: Open implementation. *IEEE Software*, (1), January 1996.

[23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In J. Knudsen, editor, *European Conference on Object-Oriented Programming*, Budapest, 2001. Springer Verlag.

[24] G. Kiczales, J. D. Rivière, and D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[25] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, March 1999. www.ccs.neu.edu/research/demeter.

[26] K. Lieberherr, D. H. Lorenz, and J. Ovlinger. Aspectual collaborations for collaboration-oriented concerns. Technical Report NU-CCS-01-08, College of Computer Science, Northeastern University, Boston, MA 02115, Nov. 2001.

[27] C. I. V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, Northeastern University, 1997. 274 pages.

[28] C. V. Lopes. Graph-based optimizations for parameter passing in remote invocations. In L.-F. Cabrera and M. Theimer, editors, *4th International Workshop on Object Orientation in Operating Systems*, pages 179–182, Lund, Sweden, August 1995. IEEE, Computer Society Press.

[29] C. V. Lopes and K. J. Lieberherr. Abstracting process-to-function relations in concurrent object-oriented applications. In R. Pareschi and M. Tokoro, editors, *European Conference on Object-Oriented Programming*, pages 81–99, Bologna, Italy, 1994. Springer Verlag, Lecture Notes in Computer Science.

[30] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New age components for old-fashioned java. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* in *Special Issue of SIGPLAN Notices*, volume 36, pages 211–222, 2001.

[31] B. Meyer. *Object-Oriented Software Construction*. Series in Computer Science. Prentice-Hall International, 1988.

[32] M. Mezini. Maintaining the consistency of class libraries during their evolution. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* in *Special Issue of SIGPLAN Notices*, 1997.

[33] M. Mezini. *Variation-Oriented Programming Beyond Classes and Inheritance*. PhD thesis, University of Siegen, 1997.

[34] M. Mezini and K. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In C. Chambers, editor, *Object-Oriented Programming Systems, Languages and Applications Conference,* in *Special Issue of SIGPLAN Notices*, number 10, pages 97–116, Vancouver, October 1998. ACM.

[35] H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal. Subject-oriented composition rules. In *Proceedings OOPSLA '95, ACM SIGPLAN Notices*, pages 235–250, Oct. 1995. Published as Proceedings OOPSLA '93, ACM SIGPLAN Notices, volume 30, number 10.

[36] D. Parnas. One the criteria to be used in decomposing systems into modules. In *Communications of the ACM*, volume 15, pages 1053–1058, 1972.

[37] M. Rinard. Aspectual programming is programming with come-from. Personal Communication to Mitch Wand, in Parking lot at NEPLS, Oct 2001.

[38] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin-layers. In *European Conference on Object-Oriented Programming*. Springer Verlag, 1998.

[39] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.

[40] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, Los Angeles, 1999. ACM.

[41] X. P. A. Team. AspectJ. Technical report, Xerox PARC, January 1999. http://www.parc.xerox.com/spl/projects/aop/.

[42] P. Wegner. The object-oriented classification paradigm. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 479–560. Cambridge, Massachusetts, MIT Press, 1988.